Seminar Series on High Performance Computing: **Introduction to Parallel Programming Techniques**

Pragnesh Patel
National Institute for Computational Sciences
pragnesh@utk.edu

Jan 28, 2014

Acknowledgement:

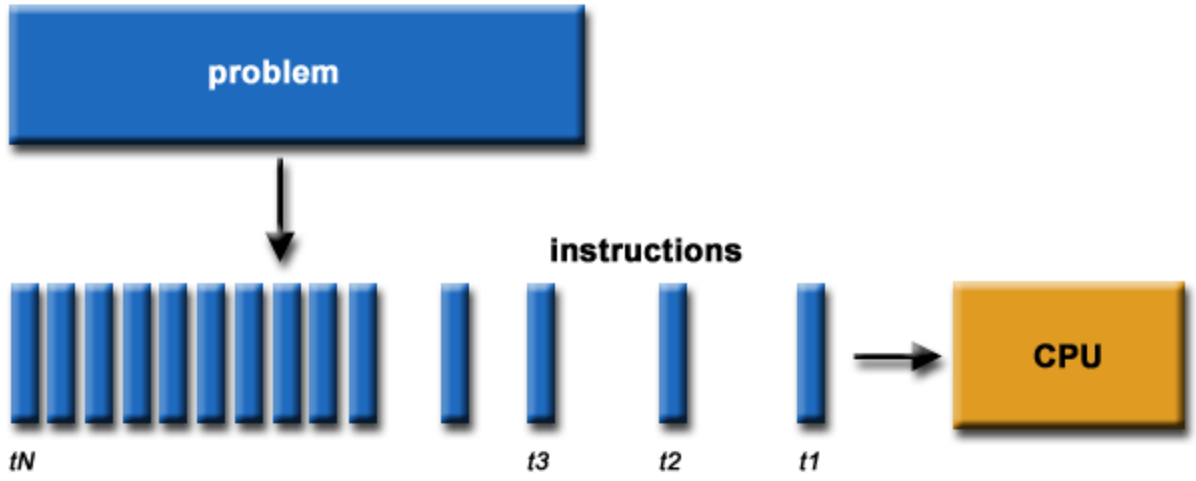
- Florent Nolot, “Introduction to Parallel Computing”, Université de Reims Champagne-Ardenne w/some modifications and augmentations by Pragnesh Patel

Overview

- The basics of parallel computing
- Parallel concepts and terminology
- Parallel memory architectures
- Programming models
- Designing Parallel Programs

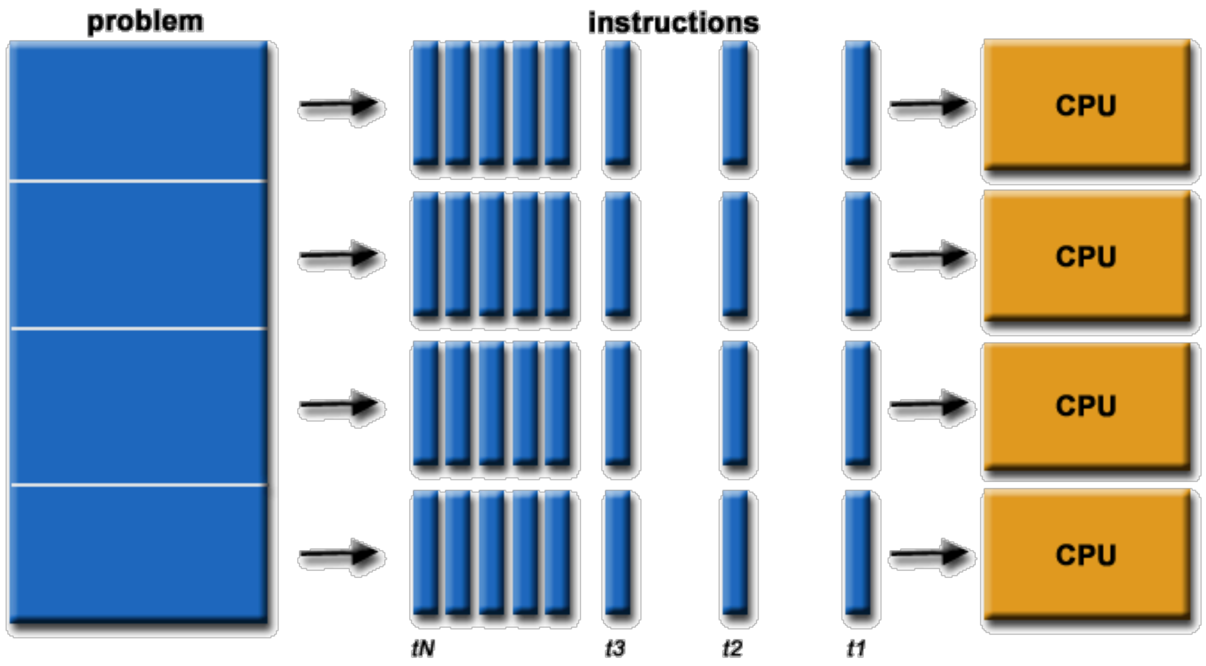
What is Parallel Computing? (1)

- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



What is Parallel Computing? (2)

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Why Parallel Computing? (1)

- This is a legitime question! Parallel computing is complex on any aspect!
- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

Why Parallel Computing? (2)

- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

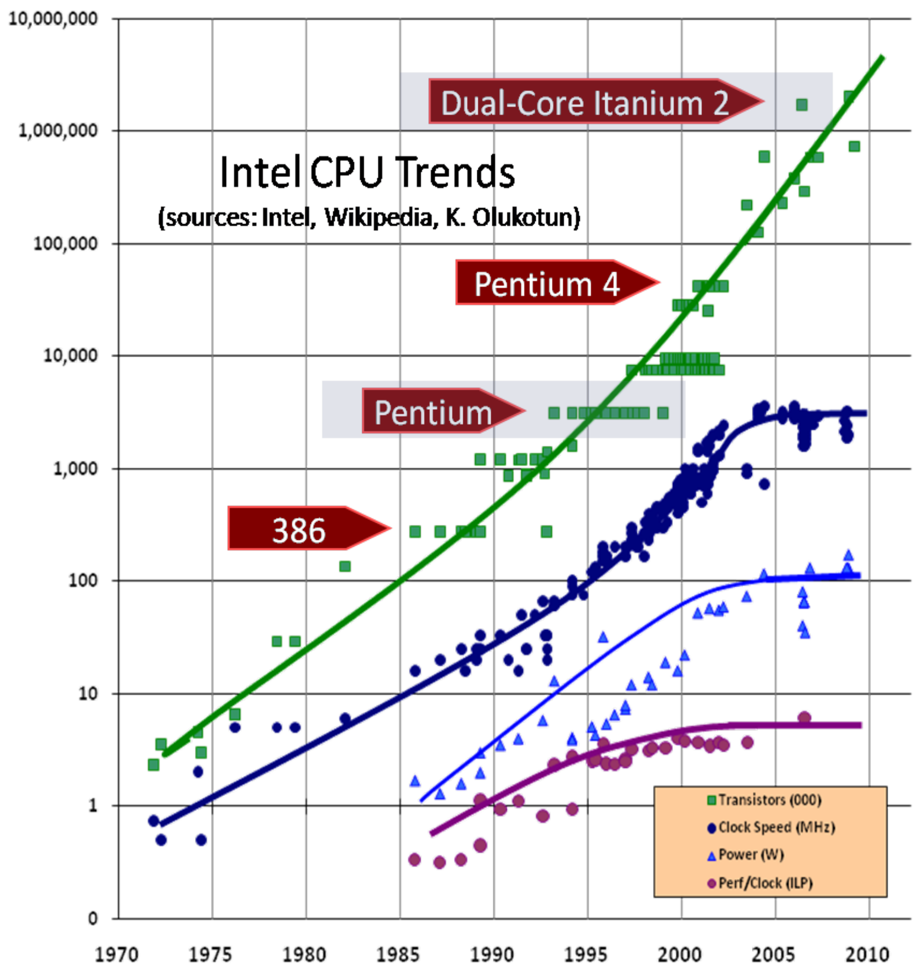
Limitations of Serial Computing

- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever faster serial computers.
- Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
- Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

Moore's Law

- The number of transistors on integrated circuits doubles approx every 2 years.
- Chip performance double every 18-24 months.
- Power consumption is proportional to frequency.
- Clock speed saturates

  at 3 to 4 GHz. (End of free lunch. Future is parallel.)



Parallel Computing: Resources

- The compute resources can include:
  - A single computer with multiple processors;
  - A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA, MIC …)
  - An arbitrary number of computers connected by a network;
  - A combination of above.

Parallel Computing: The computational problem

- The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources than with a single compute resource.

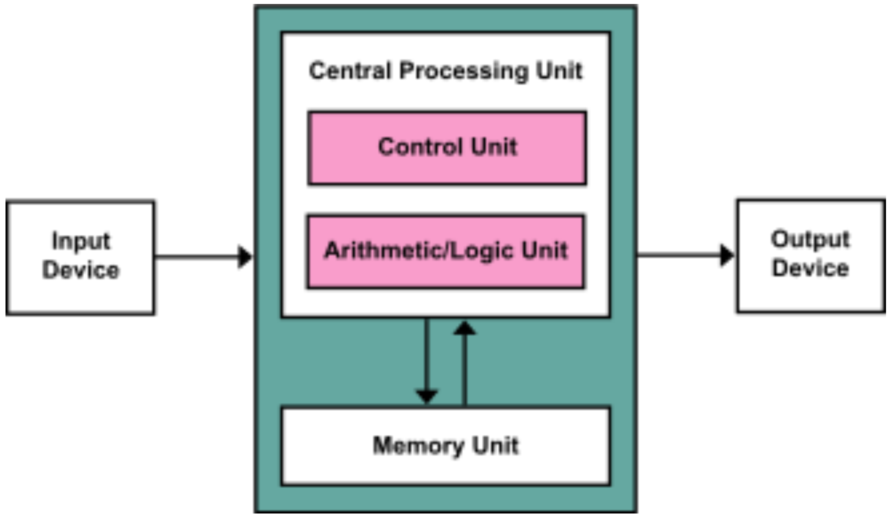
The future

- During the past 10 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.
- It will be multi-forms, mixing general purpose solutions (your PC…) and very speciliazed solutions as IBM Power, Intel MIC, GPGPU from Nvidia and others…

Concepts and Terminology

Von Neumann Architecture

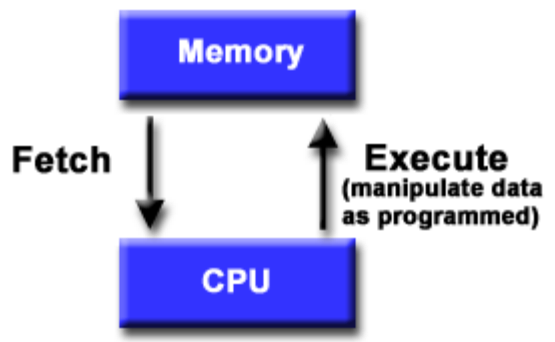
- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.



Basic Design

- Basic design
  - Memory is used to store both program and data instructions
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.

Instruction and Data

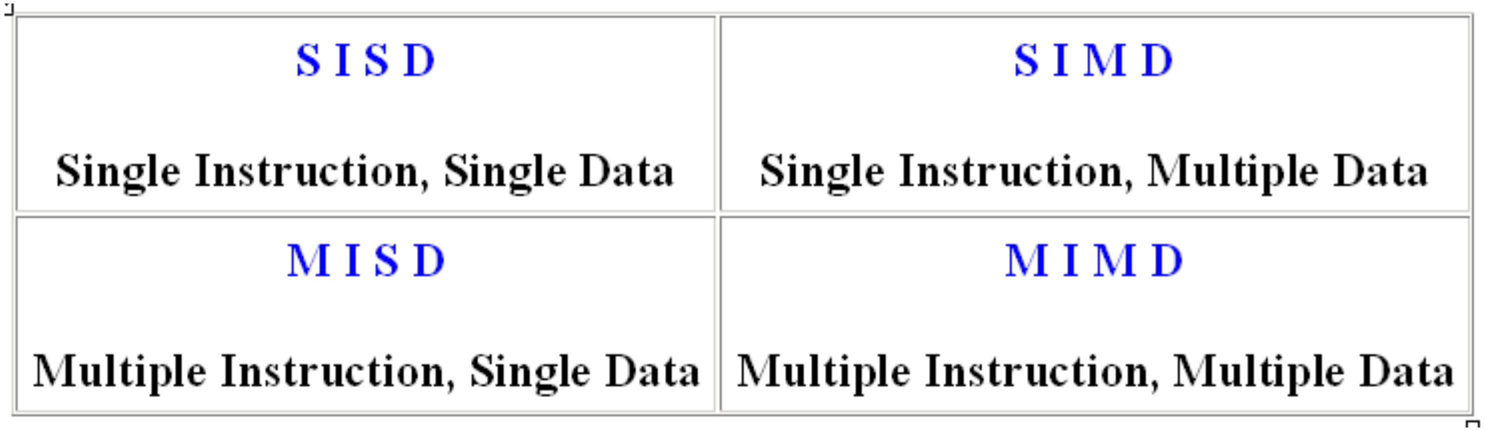


Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

Flynn Matrix

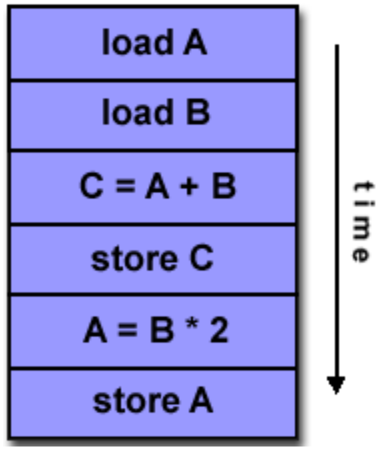
- The matrix below defines the 4 possible classifications according to Flynn



SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

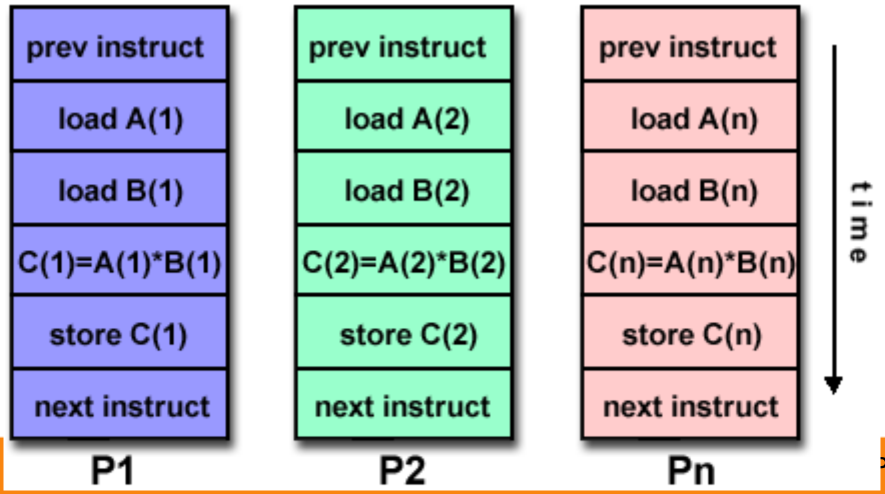
Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes



Single Instruction, Multiple Data (SIMD)

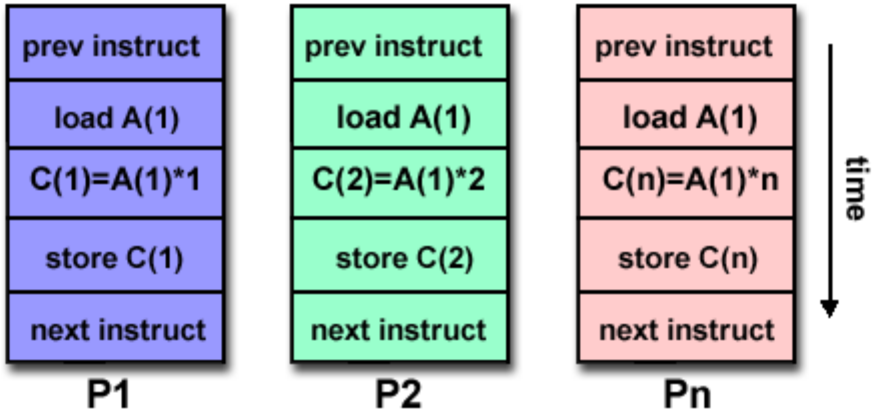
- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity,such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820



P=Processor

Multiple Instruction, Single Data (MISD)

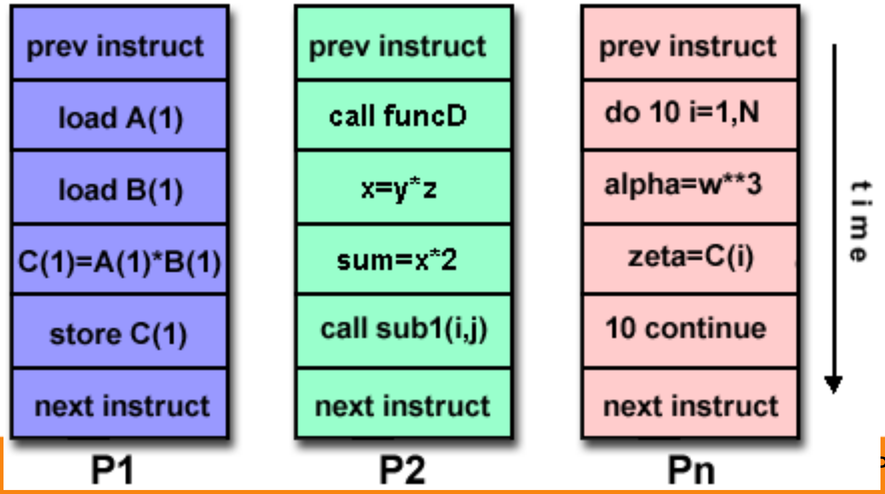
- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
- multiple cryptography algorithms attempting to crack a single coded message.



P=Processor

Multiple Instruction, Multiple Data (MIMD)

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



P=Processor

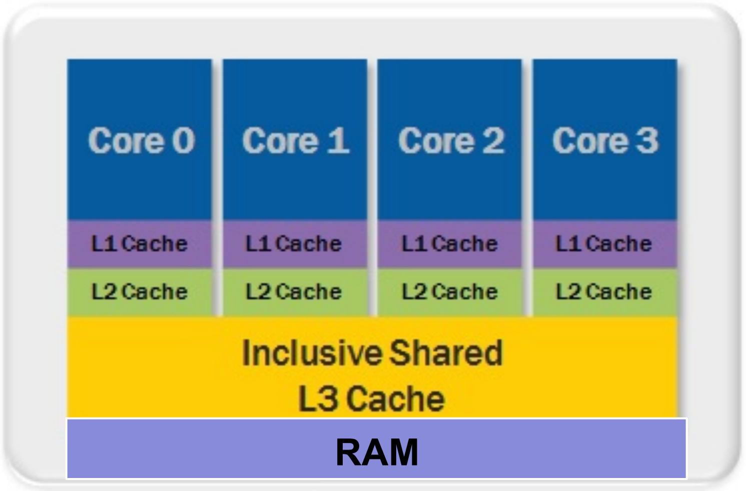
SPMD

- Another type is SPMD(*Single Program Multiple Data*), special case of MIMD.
- SPMD execution happens when a MIMD computer is programmed to have the same set of instructions per processor.
- With SPMD computers, while the processors are running the same code segment, each processor can run that code segment asynchronously.
- An example of execution on a SPMD computer.
  - One processor may take a certain branch of the if statement, and another processor may take a different branch of the same if statement.
  - Hence, even though each processor has the same set of instructions, those instructions may be evaluated in a different order from one processor to the next.

Parallel Computer Memory Architectures

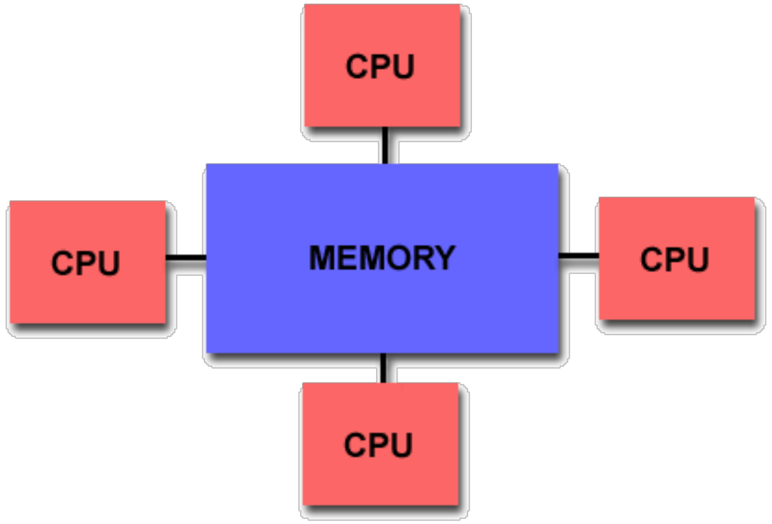
Memory architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory
- GPU Memory Model



Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as **global address space.**



Nautilus@NICS

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory : UMA vs. NUMA

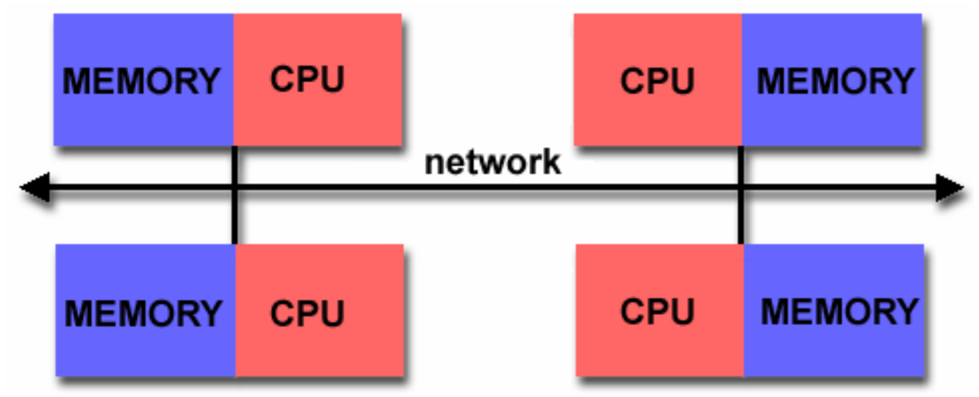
- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Non-Uniform Memory Access (NUMA):
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Shared Memory: Pro and Con

- Advantages
  - Global address space provides a **user-friendly** programming perspective to memory
  - **Data sharing** between tasks is both **fast and uniform** due to the proximity of memory to CPUs
- Disadvantages:
  - Primary disadvantage is **the lack of scalability** between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - **Programmer responsibility for synchronization** constructs that insure "correct" access of global memory.
  - **Expense**: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. No concept of global address space across all processors.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.



Distributed Memory: Pro and Con

- Advantages
  - Memory is **scalable** with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - **Cost effectiveness**: can use commodity, off-the-shelf processors and networking.
- Disadvantages
  - The **programmer is responsible for** many of the details associated with **data communication** between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - **Non-uniform memory access (NUMA) times**

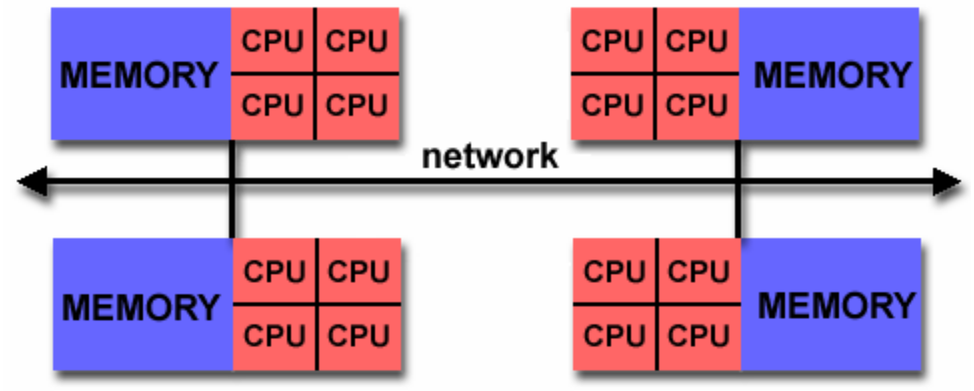
Hybrid Distributed-Shared Memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Hybrid Distributed-Shared Memory

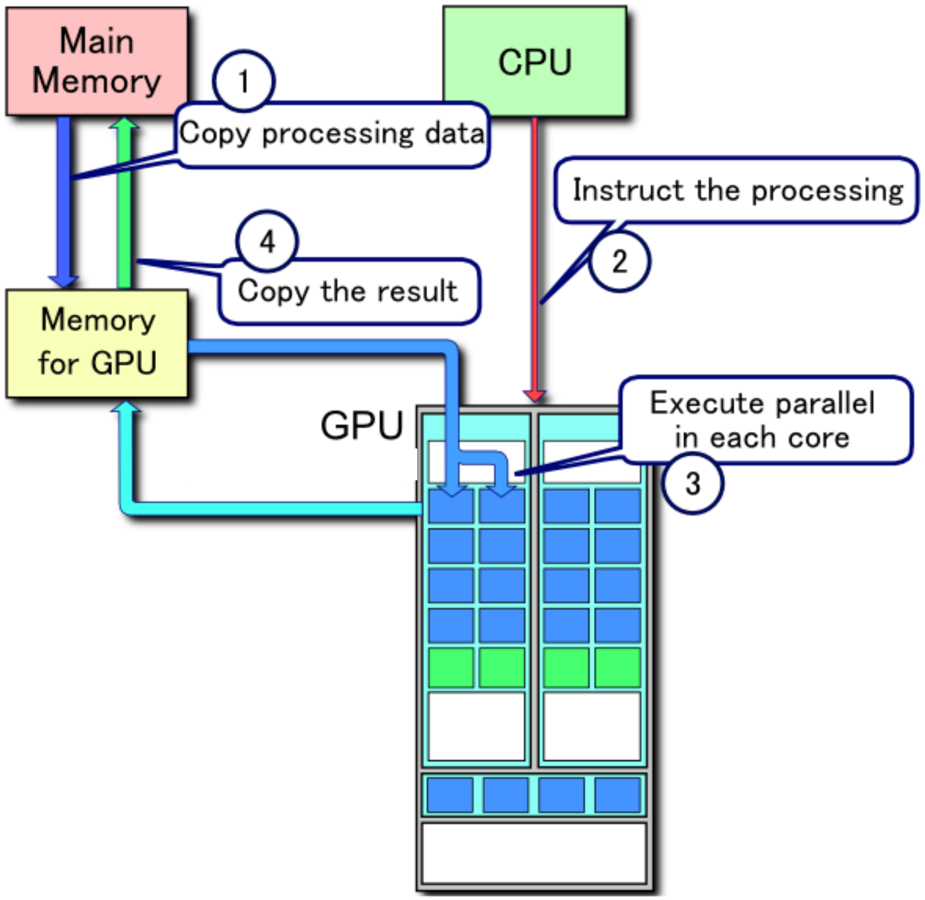
- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



Kraken@NICS

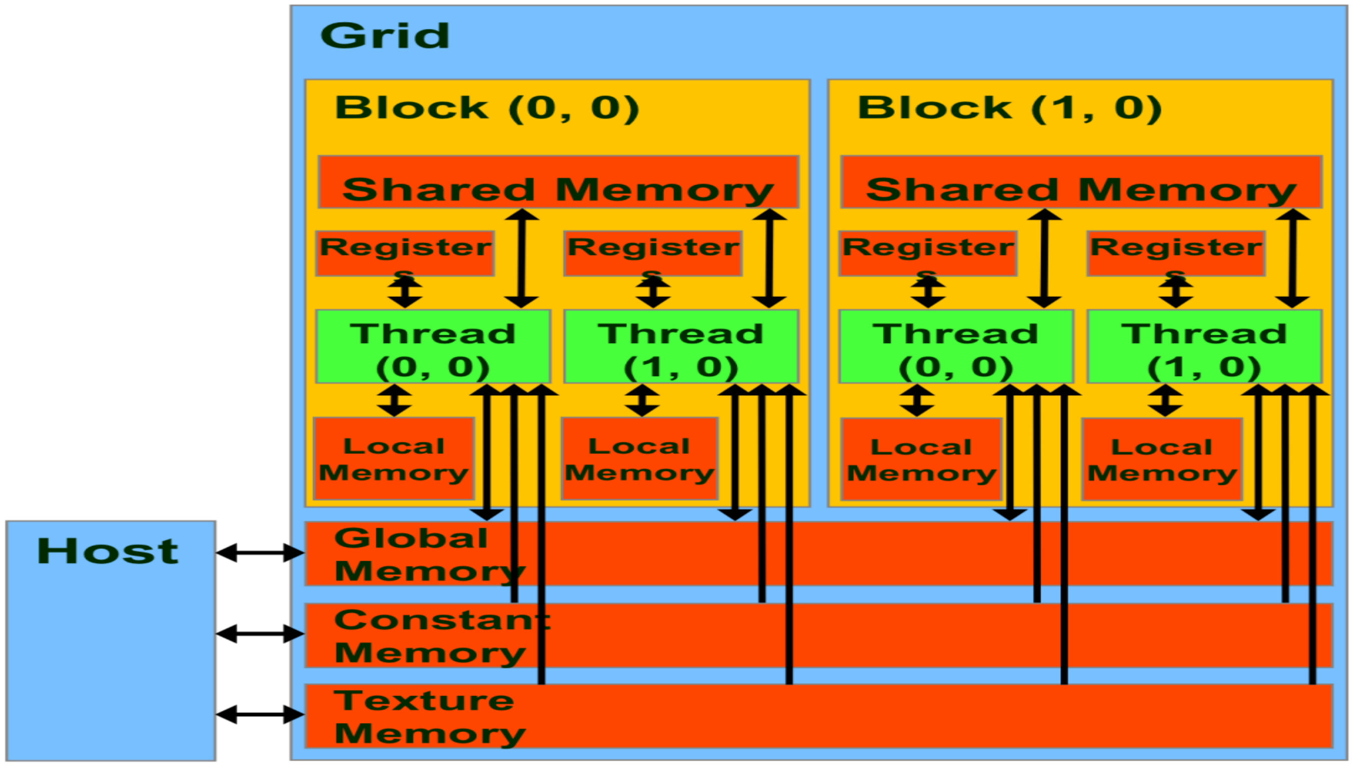
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- Network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

CPU-GPU together



GPU memory model

- GPU has much more aggressive memory model.



Parallel Programming Models

Parallel programming models

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Task Parallel Model
- Other Models

Overview

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Task Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.

Overview

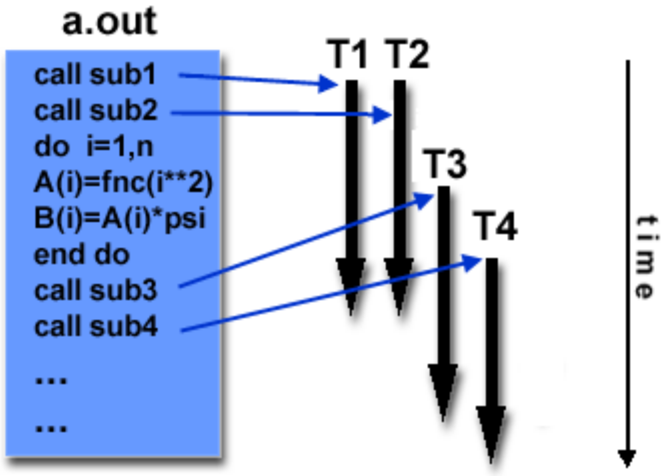
- Although it might not seem apparent, these models are NOT specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware.
- Which model to use is often a combination of what is available and personal choice. **There is no "best" model**, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space(Global Address Space), which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

Shared memory architecture ?

Threads Model



- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
  - The main program **a.out** is scheduled to run by the native operating system. a.out loads and acquires all of the necessary system and user resources to run.
  - a.out performs some serial work, and then **creates a number of tasks (threads)** that can be scheduled and run by the operating system concurrently.
  - **Each thread has local data**, but also, **shares the entire resources of a.out**. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of a.out.
  - A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - **Threads communicate** with each other **through global memory** (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
  - Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed.
- Threads are commonly **associated with shared memory architectures** and **operating systems**.

Threads Model Implementations

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives embedded in either serial or parallel source code
- In both cases, the programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.
- **POSIX Threads**
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - Commonly referred to as Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
  - Very explicit parallelism; requires significant programmer attention to detail.

Threads Model: OpenMP

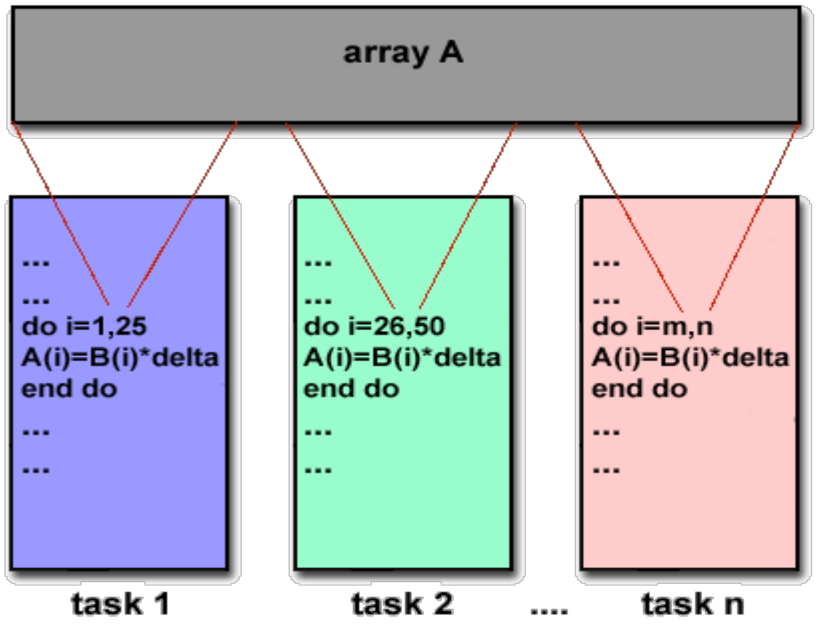
- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

Message Passing Model

- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Message Passing Model Implementations: MPI

- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. A few offer a full implementation of MPI-2.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.



Data Parallelism

- The same code segment runs concurrently on each processor, but each processor is assigned its own part of the data to work on.
  - Do loops (in Fortran) define the parallelism.
  - The iterations must be independent of each other.
  - Data parallelism is called "fine grain parallelism" because the computational work is spread into many small subtasks.
- Example
  - Dense linear algebra, such as matrix multiplication, is a perfect candidate for data parallelism.

Data Parallelism: Example

Original Sequential Code

```
DO K=1,N
DO J=1,N
DO I=1,N
C(I,J) = C(I,J) +
A(I,K)*B(K,J)
END DO
END DO
END DO
```

Parallel Code

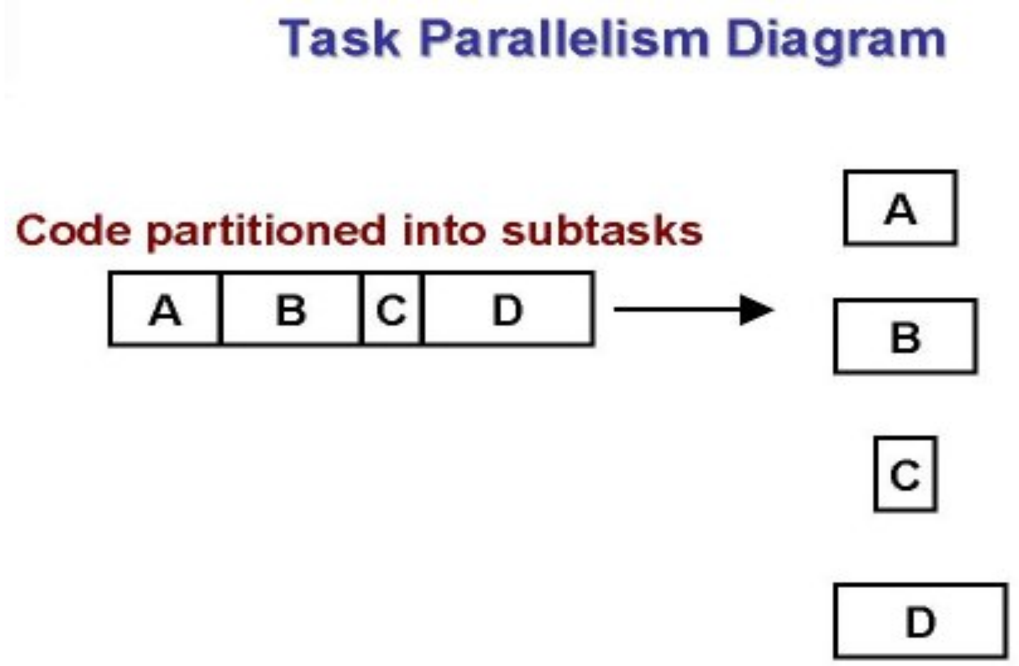
```
!$OMP PARALLEL DO
DO K=1,N
DO J=1,N
DO I=1,N
C(I,J) = C(I,J) +
A(I,K)*B(K,J)
END DO
END DO
END DO
!$END PARALLEL DO
```

Task Parallelism

- Task parallelism may be thought of as the opposite of data parallelism.
- Instead of the same operations being performed on different parts of the data, each process performs different operations.
- You can use task parallelism when your program can be split into independent pieces, often subroutines, that can be assigned to different processors and run concurrently.
- Task parallelism is called "coarse grain" parallelism because the computational work is spread into just a few subtasks.
- More code is run in parallel because the parallelism is implemented at a higher level than in data parallelism.
- Task parallelism is often easier to implement and has less overhead than data parallelism.

Task Parallelism

- The abstract code shown in the diagram is decomposed into 4 independent code segments that are labeled A, B, C, and D. The right hand side of the diagram illustrates the 4 code segments running concurrently.



Task Parallelism: Example

Original Code

```
program main


code segment labeled A

code segment labeled B

code segment labeled C

code segment labeled D


end
```

Parallel Code

```
program main
!$OMP PARALLEL
!$OMP SECTIONS
code segment labeled A
!$OMP SECTION
code segment labeled B
!$OMP SECTION
code segment labeled C
!$OMP SECTION
code segment labeled D
!$OMP END SECTIONS
!$OMP END PARALLEL
end
```

OpenMP Style of Parallelism

- can be done incrementally as follows:
  - Parallelize the most computationally intensive loop.
  - Compute performance of the code.
  - If performance is not satisfactory, parallelize another loop.
  - Repeat steps 2 and 3 as many times as needed.
- The ability to perform *incremental parallelism* is considered a positive feature of data parallelism.

Other Models

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

Hybrid

- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing

Single Program Multiple Data (SPMD)

- Single Program Multiple Data (SPMD):
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data

Multiple Program Multiple Data (MPMD)

- Multiple Program Multiple Data (MPMD):
- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data

Designing Parallel Programs

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Agenda

- Designing and developing parallel programs has characteristically been a very manual process. The programmer is typically responsible for both identifying and actually implementing parallelism.
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs. The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

- A parallelizing compiler generally works in two different ways:
  - Fully Automatic
    - The compiler analyzes the source code and identifies opportunities for parallelism.
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - Programmer Directed
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.

- If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer. However, there are several important caveats that apply to automatic parallelization:
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex
  - Most automatic parallelization tools are for Fortran
- The remainder of this section applies to the manual method of developing parallel codes.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

Example of Parallelizable Problem

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

- This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

Example of a Non-parallelizable Problem

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

```
F(k + 2) = F(k + 1) + F(k)
```

- This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the k + 2 value uses those of both k + 1 and k. These three terms cannot be calculated independently and therefore, not in parallel.

Identify the program's ***hotspots***

- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- **Profilers** and **performance analysis tools** can help here
- Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Identify ***bottlenecks*** in the program

- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

Other considerations

- Identify inhibitors to parallelism. One common class of inhibitor is ***data dependence***, as demonstrated by the Fibonacci sequence above.
- **Investigate other algorithms if possible**. This may be the single most important consideration when designing a parallel application.

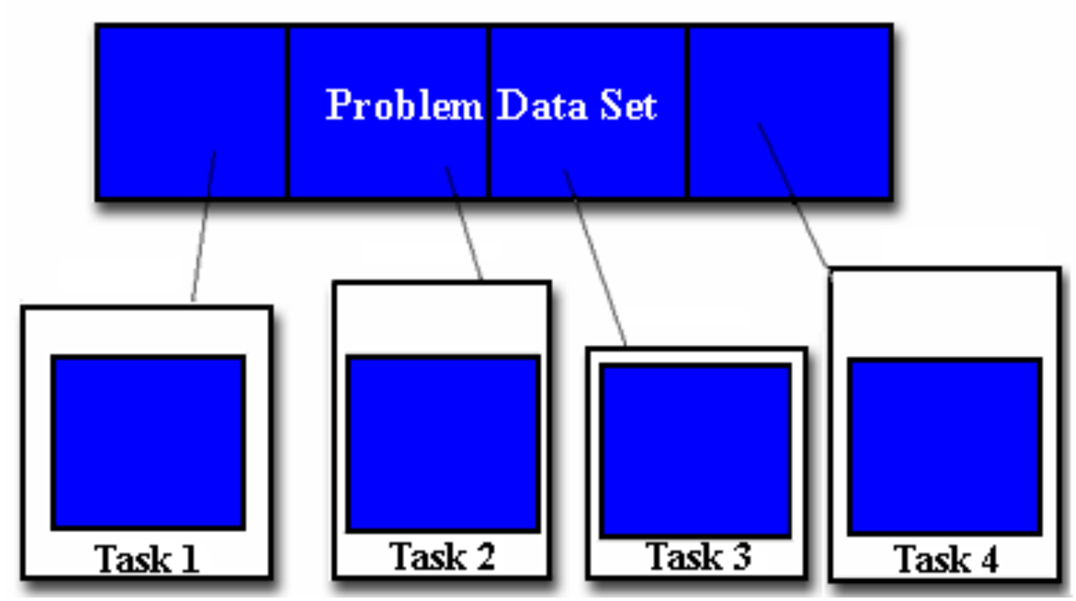
Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.
- There are two basic ways to partition computational work among parallel tasks:
  - ***domain decomposition***
    and
  - ***functional decomposition***

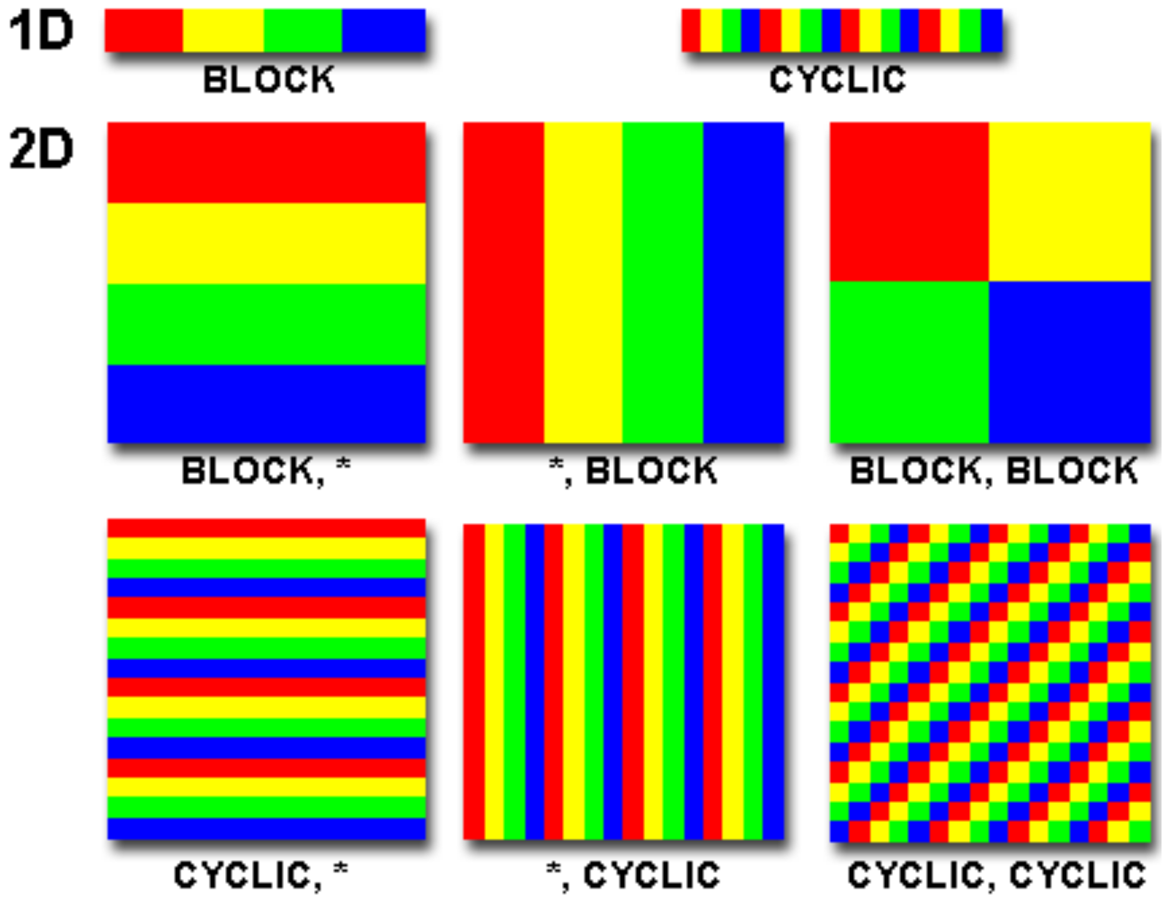
Domain Decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.



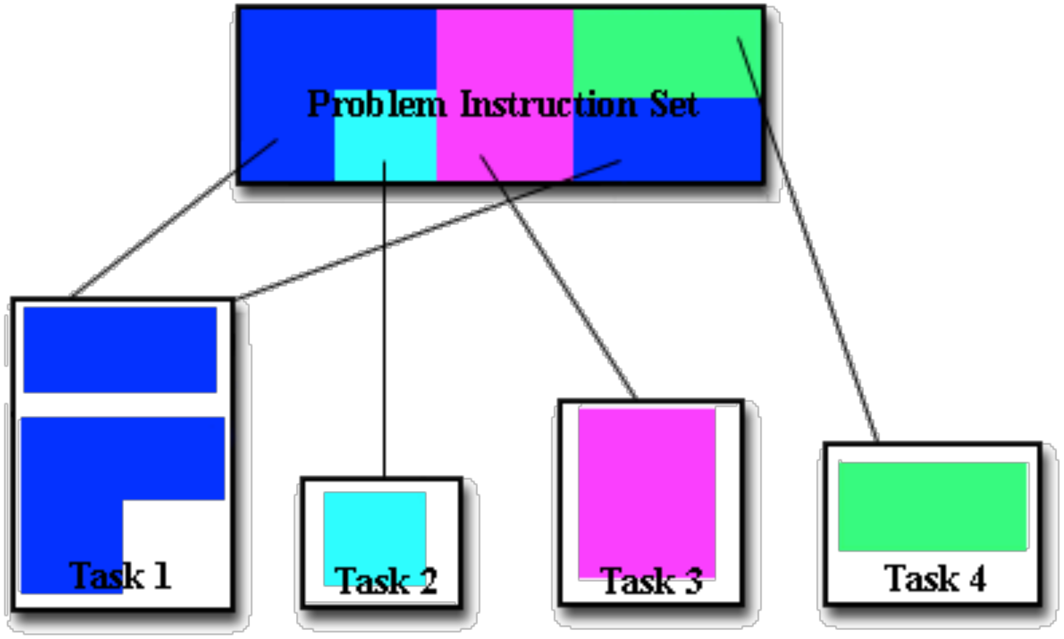
Partitioning Data

- There are different ways to partition data



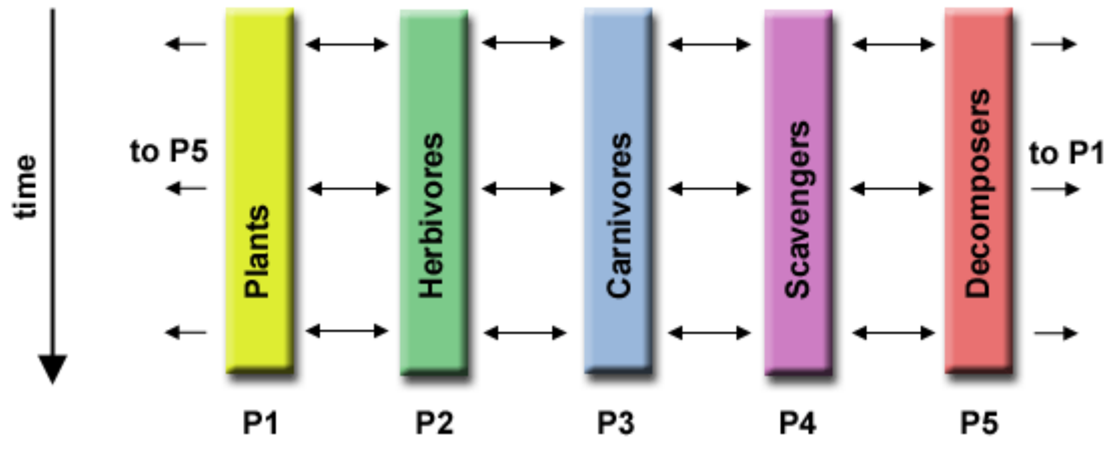
Functional Decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks. For example
  - Ecosystem Modeling
  - Signal Processing
  - Climate Modeling



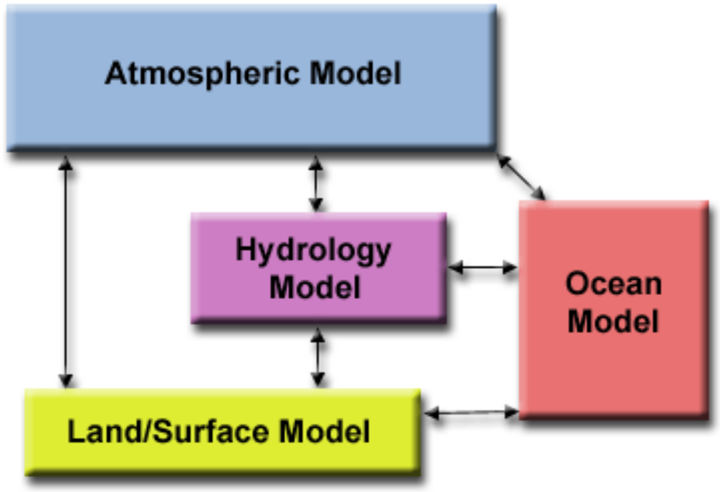
Ecosystem Modeling

- Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



Climate Modeling

- Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



- Combining these two types of problem decomposition is common and natural.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Who Needs Communications?

- The need for communications between tasks depends upon your problem
- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.
- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

Factors to Consider (1)

- There are a number of important factors to consider when designing your program's inter-task communications
- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Factors to Consider (2)

- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
  - ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Factors to Consider (3)

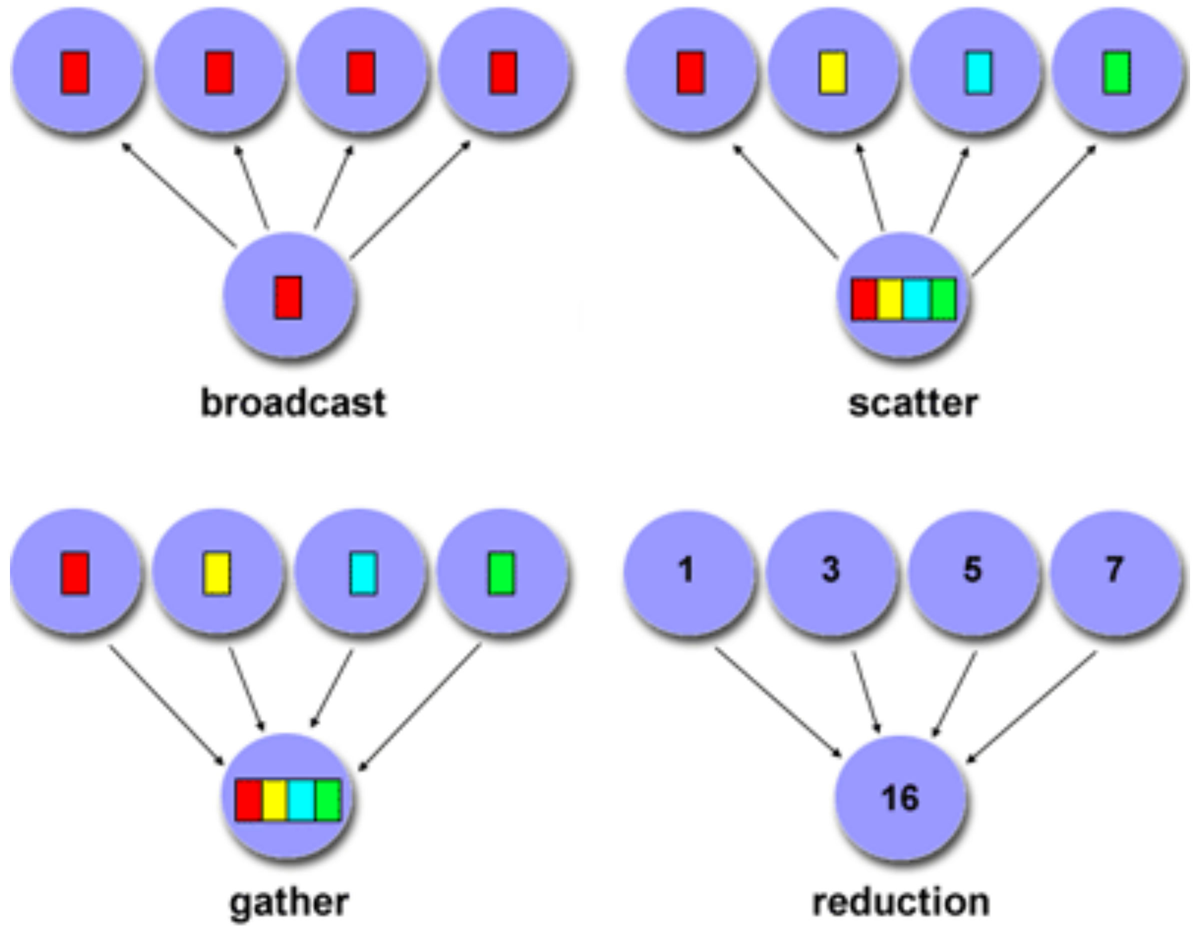
- **Synchronous vs. asynchronous communications**
  - Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
  - Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Factors to Consider (4)

- **Scope of communications**
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
  - ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

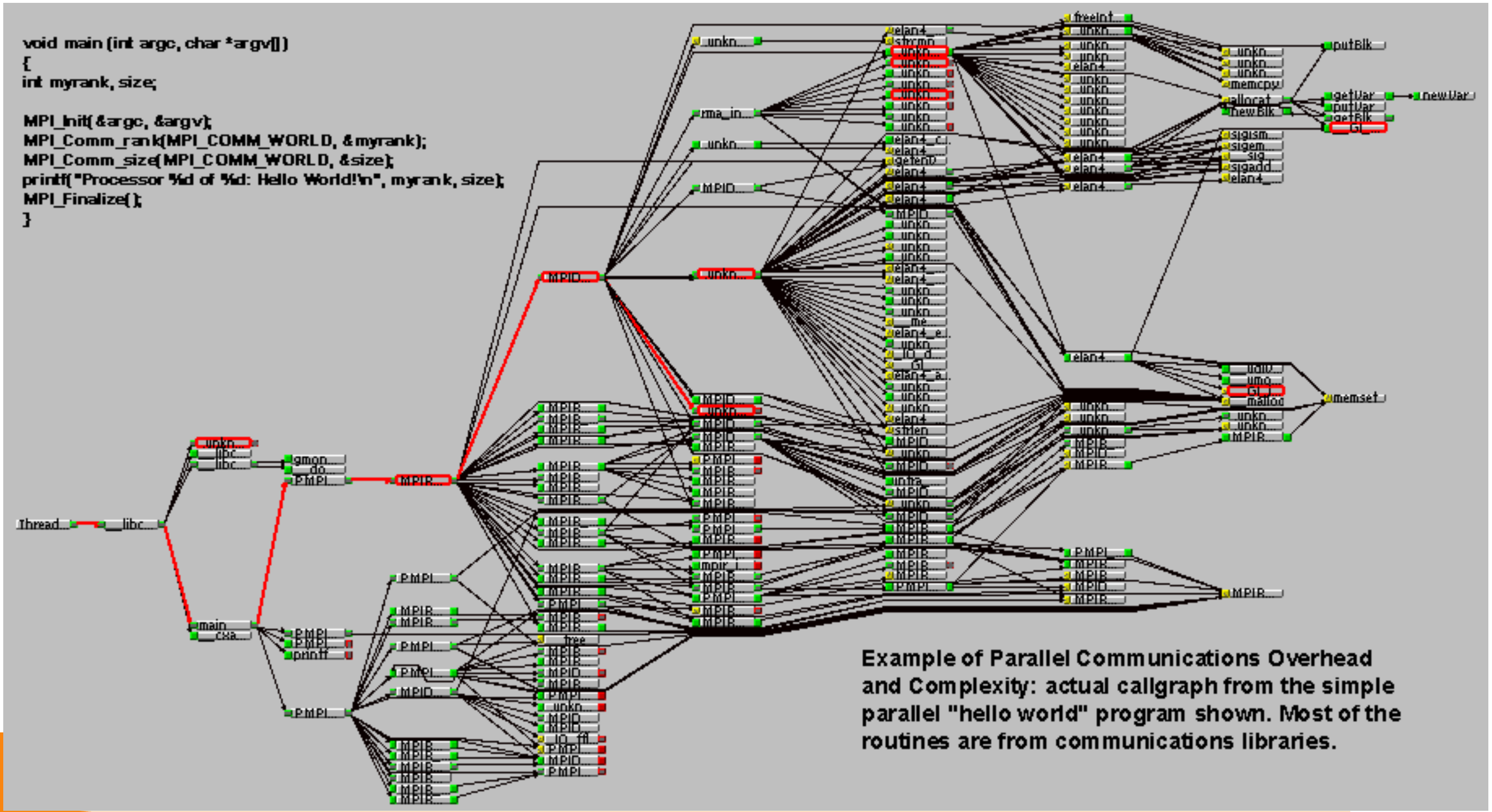
Collective Communications

- **Examples**



Factors to Consider (5)

- **Overhead and Complexity**



```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize();
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

Factors to Consider (6)

- **Finally, realize that this is only a partial list of things to consider!!!**

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Types of Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.
- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking
- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
- A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

Examples (1): Loop carried data dependence

```
      DO 500 J = MYSTART,MYEND
           A(J) = A(J-1) * 2.0500
      CONTINUE
```

- The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

Examples (2): **Loop independent data dependence**

```
task 1          task 2
------          ------
X = 2           X = 4
.               .
.               .
Y = X**2        Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.

How to Handle Data Dependencies?

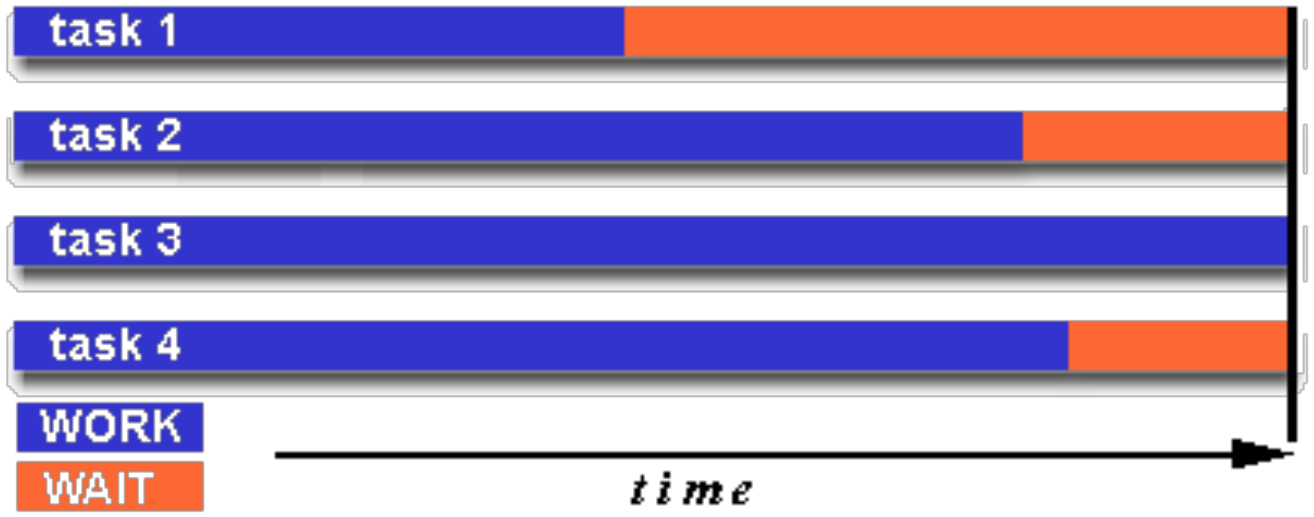
- Distributed memory architectures - communicate required data at synchronization points.
- Shared memory architectures -synchronize read/write operations between tasks.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definition

- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



How to Achieve Load Balance? (1)

- **Equally partition the work each task receives**
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

How to Achieve Load Balance? (2)

- **Use dynamic work assignment**
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
  - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Agenda

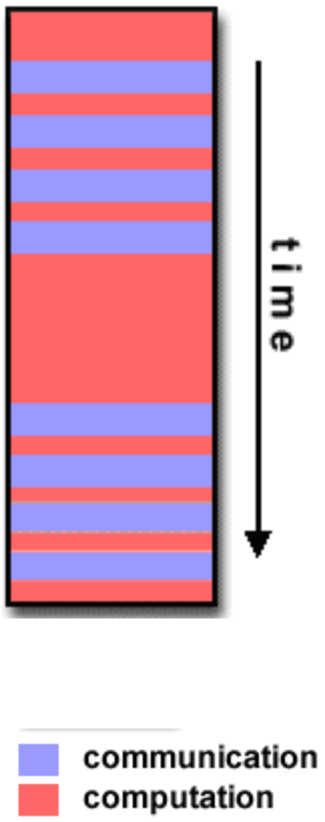
- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- Computation / Communication Ratio:
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.
- Fine grain parallelism
- Coarse grain parallelism

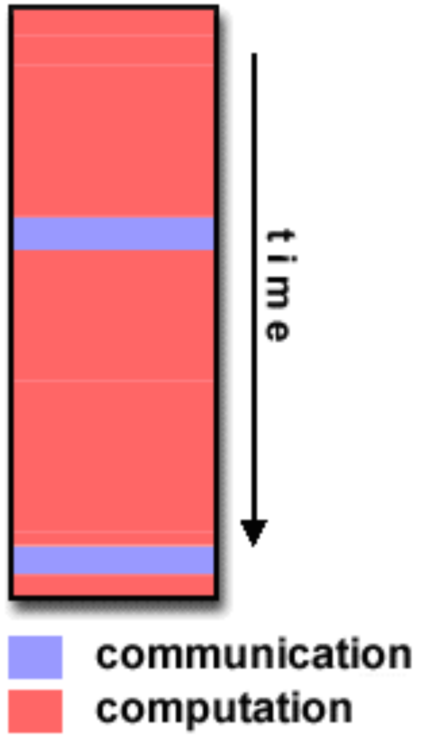
Fine-grain Parallelism

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



Coarse-grain Parallelism

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently



Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

The bad News

- I/O operations are generally regarded as inhibitors to parallelism
- Parallel I/O systems are immature or not available for all platforms
- In an environment where all tasks see the same filespace, write operations will result in file overwriting
- Read operations will be affected by the fileserver's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks

The good News

- Some parallel file systems are available. For example:
  - GPFS: General Parallel File System for AIX (IBM)
  - Lustre: for Linux clusters (Cluster File Systems, Inc.)
  - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/Argonne/Ohio State/others)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - HP SFS: HP StorageWorks Scalable File Share. Lustre based parallel file system (Global File System for Linux) product from HP
- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

Some Options

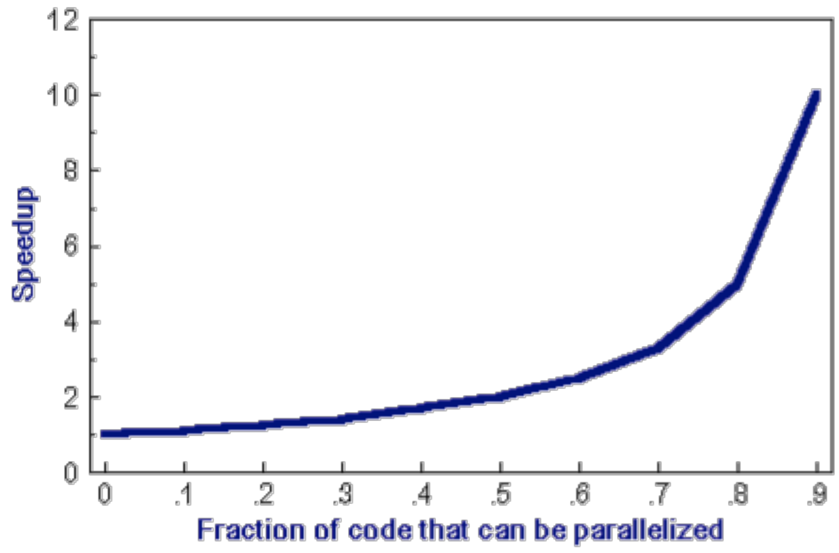
- If you have access to a parallel file system, investigate using it. If you don't, keep reading...
- Rule #1: Reduce overall I/O as much as possible
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- For distributed memory systems with shared filespace, perform I/O in local, non-shared filespace. For example, each processor may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.
- Create unique filenames for each tasks' input/output file(s)

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Amdahl's Law

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:



```
                         1
speedup    =     ---------
                   1  - P
```

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Amdahl's Law

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by

```
                             1
speedup    =    ------------
                   P    +  S
                  ---
                   N
```

- where P = parallel fraction, N = number of processors and S = serial fraction

Amdahl's Law

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable)

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Amdahl's Law

- However, certain problems demonstrate increased performance by increasing the problem size. For example:

2D Grid Calculations	**85 seconds**	**85%**
Serial fraction	**15 seconds**	**15%**

- We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

2D Grid Calculations	**680 seconds**	**97.84%**
Serial fraction	**15 seconds**	**2.16%**

- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Complexity

- In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance
- Adhering to "good" software development practices is essential when when working with parallel applications - especially if somebody besides you will have to work with the software.

Portability

- Thanks to standardization in several APIs, such as MPI, POSIX threads, HPF and OpenMP, portability issues with parallel programs are not as serious as in years past. However...
- All of the usual portability issues associated with serial programs apply to parallel programs. For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem.
- Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability.
- Operating systems can play a key role in code portability issues.
- Hardware architectures are characteristically highly variable and can affect portability.
- It is becoming hard with GPU, MIC and FPGA architectures.

Resource Requirements

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Scalability

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- As with debugging, monitoring and analyzing parallel program execution is significantly more of a challenge than for serial programs.
- A number of parallel tools for execution monitoring and program analysis are available.
- Some are quite useful; some are cross-platform also.
- One starting point: Performance Analysis Tools Tutorial
- Work remains to be done, particularly in the area of scalability.

References

- http://www. cosy.univ-reims.fr/~fnolot/.../ introduction_to_parallel_computing.ppt
- http://www.intel.com/
- http://www.wikipedia.org/
- http://www.nvidia.com/

Thank you Daniel for helpful suggestions!!!